Apple II
Technical Notes

Developer Technical Support

Apple IIGS
#75: BeginUpdate Anomaly

Revised by: Dave Lyons May 1992
Written by: Eric Soldan January 1990

This Technical Note discusses a Window Manager anomaly with the handling of the `visRgn` and the `updateRgn` between `BeginUpdate` and `EndUpdate` calls.
Changes since January 1990: Updated for System 6.0. `CopyPixels` is in a static segment, and GS/OS automatically prompts for disks on the text screen when necessary to avoid interfering with a window update in progress.

If an application calls `BeginUpdate`, it needs to be fully aware of what is going on behind the scenes in terms of its `visRgn` and `updateRgn`. Typically an application has `TaskMaster` handle the update events. `TaskMaster` calls `BeginUpdate`, the application update procedure, then `EndUpdate`. So any application that uses `TaskMaster` to handle updates, whether or not it makes any `BeginUpdate` calls directly, needs to be aware of problem described in this Note.

`BeginUpdate` is responsible for intersecting the `visRgn` and the `updateRgn` and making the intersection of these two regions the temporary `visRgn`. (`EndUpdate` undoes this effect.) Following are the steps `BeginUpdate` takes to do this:

1. Localize the `updateRgn`. (All `grafPort` regions are local, therefore the `visRgn` is local. All window regions are global, therefore the `updateRgn` is global. One of them has to change if they are to be intersected correctly.)
2. Intersect the `visRgn` and localized `updateRgn`, then place the result in the `updateRgn`.
3. Swap the `visRgn` and `updateRgn` handles.

   The handle swapping has two effects:

   - Makes the intersection region the current `visRgn`.
   - Saves the real `visRgn` as the `updateRgn`. (Saving the real `visRgn` is necessary because everything has to be restored to normal by `EndUpdate`.)

`EndUpdate` restores things to normal after an update procedure is finished. When an application calls `EndUpdate`, it swaps back the handles and sets the `updateRgn` to empty.

So What's the Problem?

The problem is that the `updateRgn` is not a very good place to save the `visRgn`. Since `InvalRect` and `InvalRgn` modify the `updateRgn`, if either of these two calls is made between a `BeginUpdate` and `EndUpdate`, they modify the saved `visRgn`. When the update is finished, `EndUpdate` restores the modified `visRgn` instead of the original.

The solution to this problem seems simple enough: don't call `InvalRect` or `InvalRgn` between `BeginUpdate` and `EndUpdate`. Unfortunately, there are other calls which can call `BeginUpdate`, `EndUpdate`, `InvalRect`, and `InvalRgn`, so an application might inadvertently call one of these routines.

If this situation isn't bad enough already, you could really mess things up by opening another window between `BeginUpdate` and `EndUpdate` calls. Opening a window at this time may seem like a perfectly normal thing (i.e., to display an alert); however, opening a window forces the recalculation of the `visRgn` for any windows obscured by the new window. If the window being updated has its `visRgn` recalculated, the application obviously loses the `visRgn` that `BeginUpdate` created. This doesn't seem too serious since the `visRgn` is restored to the entire visible part of the window when the new window is closed; however, it does mean that the application would have to update the entire window instead of the original `updateRgn`.

Unfortunately, the Window Manager also posts update events for the portion of the window that was obscured, and it does this by changing the `updateRgn`. Of course the `updateRgn` for the window being updated is really the `visRgn` that is being "safely" preserved until the `EndUpdate` call. So, there are some really good reasons why this can't be done.

Okay, so along with not making calls to `InvalRect` and `InvalRgn` between `BeginUpdate` and `EndUpdate`, an application cannot open any other windows either. Good.

Now to make things even worse.

Starting with System 5.0, some toolbox functions are stored on disk in dynamic segments and loaded when they are first called. For example, `CopyPixels` is in a dynamic segment in System versions 5.0 through 5.0.3. If the startup disk is not available and the system prompts for it between `BeginUpdate` and `EndUpdate` by calling `AlertWindow`, the bad things discussed above happen.

Starting with System 6.0, the system is smart enough not to prompt for a disk using `AlertWindow` if a window update is in progress. (Internally, GS/OS calls `WindStatus` to see if it can prompt on the graphics screen. If `BeginUpdate` has been called more times than `EndUpdate`, `WindStatus` fibs by returning with the carry set. GS/OS takes the hint and prompts for the disk with a text dialog instead.)

But I Have to Do…

If you absolutely must do some of the things previously discussed, there is a way to accomplish it. It is not simple, but it can be done.

Assuming that `BeginUpdate` has been called, and an application is in its update procedure:

1. Create a new region and copy the `visRgn` into it. Doing this allows the application to restore the `visRgn` to just the area to be updated that `BeginUpdate` calculated. This needs to be done for any other windows which obscure a part the the window being updated. Again, these are not windows that an application would open directly. `CopyPixels` may open a window, since it is a dynamic segment and may need to get loaded from a disk that is off-line.
2. Create a new region, then swap its handle with the `updateRgn` handle. This protects the real `visRgn` and lets an application call `InvalRect` and `InvalRgn` at any time if necessary. It also means the application doesn't need to worry about the system making these calls either. The `updateRgn` is also an empty region after the swap, so any contributions to it constitute a valid update event that needs to be handled.
3. Do the update part of the update procedure. In this part, if the application has any calls to `CopyPixels`, or any other QuickDraw Auxiliary dynamic segment functions, after the call is completed, copy the saved `visRgn` back to the `visRgn` of the `grafPort`. The closing of the dynamic segment alert window recalculates the `visRgn`, and copying it undoes this effect. Do **not** do the same for the `updateRgn`. Leave the `updateRgn` alone. We are accumulating an actual `updateRgn`, and the closing of the alert window for the dynamic segment may have contributed to this region.

There are two methods for leaving the update procedure. Although the second method works whether or not an application uses `TaskMaster`, if an application does not use `TaskMaster`, then the first method is simpler.

The procedure without using `TaskMaster` (i.e., you made the `BeginUpdate` call, and you will make the `EndUpdate` call) is as follows:

A. Dispose of the region created in Step 1. This region was only needed to restore the partial `visRgn` that `BeginUpdate` calculated after a window was opened.
B. Swap the `updateRgn` handle with the region handle created in Step 2.
C. Make the `EndUpdate` call.
D. If the region created in Step 2 is not empty, copy this region into the `updateRgn` for the window with `CopyRgn`. You can't just do an `InvalRgn` with it because `InvalRgn` globalizes the region and the region is already global. You want to copy the region since this generates a valid update event. You can use `CopyRgn` instead of `UnionRgn` because the update region is empty.
E. Dispose of the region created in Step 2.

With `TaskMaster`, things are a little messier. Since `TaskMaster` makes the `EndUpdate` call, you have less control over the situation. It is important to do the `EndUpdate` before generating the update event. Posting the update event has to happen outside the update procedure, since you have to leave the update procedure for `TaskMaster` to do the `EndUpdate`. So it follows that you do Steps A and B, post an application event to handle the rest externally, and when the application event is handled, do Steps D and E.

Some consideration was given to posting an application event via the `PostEvent` call. Unfortunately, there is a possibility that this application event will drop out of the queue not handled. When the queue is full, the oldest event is dropped, and this could occur to application events, which would be very bad in this case. Due to this possibility, posting an application event refers to setting a global variable that is checked before the `TaskMaster` call in the main event loop. This can be considered equivalent to posting an event via the `PostEvent` call.

So, the `TaskMaster` case would be as follows:

A. Dispose of the region created in Step 1.
B. Swap the `updateRgn` handle with the region handle created in Step 2.
C. Store the handle of the region created in Step 2 in a global variable named `eventUpdateRgn`. Store the current window port in a global variable named `eventWindowPort`.
D. Return to `TaskMaster`, which returns to the main event loop.
E. Immediately after the `TaskMaster` call in the main event loop, check the global variable `eventUpdateRgn`. If it is not `NULL` then:
    a. Copy the region into the `updateRgn` of the window `eventWindowPort`. Using `CopyRgn` is the easiest way to copy the region. (Copying the region posts an update event if the event `UpdateRgn` is not `NULL`.
    b. Dispose of the region `eventUpdateRgn`, then set the variable `eventUpdateRgn` to `NULL`, so that this “event” won’t be handled again.

Of course, the simplest way to handle all of this is to avoid situations where you have to take the steps described above. If things like opening a window (or allowing the system to open one) and `InvalRect` and `InvalRgn` can be avoided between calls to `BeginUpdate` and `EndUpdate`, so can all of this ugliness.

Further Reference

- *Apple IIGS Toolbox Reference,* Volume 2